Middleware in robotics
Simone Ceriani, Martino Migliavacca

Abstract—Robotic systems require significant interaction and coordination of hardware and software elements. In the context of software engineering, the concept of middleware earned a very strong role in the entire software development process. In robotic applications the use of a middleware can help improving the organization, the maintainability and the efficiency of the code that controls the robot. In this work we want to give an overview of the most used middleware frameworks for robotics, identifying the common basic concepts and goals and highlighting the lacks, in our opinion, of the current projects. We also propose a different view of a robotic system, as a distributed system of cooperating devices, that needs an approach different from the one used in computer science, from which the concept of middleware is inherited.

1 INTRODUCTION

Robotic systems require significant interaction and coordination of hardware and software elements. The aggregation of a large number of sub-systems, even within a single robot, requires careful consideration of architectural issues to promote flexibility, performance and reuse of both hardware and software. These issues are especially important when we consider to use many different kinds of robots and embedded systems, types of applications, models of programming, and operating environments.

In the context of software engineering, the concept of middleware earned a very strong role in the entire software development process. Middleware is not only a set of libraries and APIs for specific applications. It represents a way of thinking the software, which will be designed and developed on a specific middleware, that defines the foundation for a complete application.

People involved in robotics are getting conscious that developing multirobot systems might require tools for supporting the developers to deal with organization, integration and communication issues [1].

2 MIDDLEWARE

2.1 Middleware origins
Middleware is a relatively new addition to computer science concepts. It gained popularity in the 1980s as a solution to wrap legacy system and interface them with newer application. Although its recent diffusion, the introduction of the term “middleware” can be dated to 1968 [3], when d’Agapeyeff used it for the first time to identify that part of software that could be shared between different applications. In his original idea, d’Agapeyeff described an inverse pyramid (Fig. 1) that describes a primitive software architecture in which the high level applications are build up using a set of middle level routines (middleware). This, in turn, is based on a small set of service routines.

2.2 What is middleware?
Middleware is computer software that connects software components or applications. Gener-
ally a middleware consists of a set of services that allows multiple processes running on one or more machines to interact. This technology evolved to provide interoperability in support the diffusion of coherent distributed architectures, which are most often used to support and simplify the development of complex distributed applications. It includes web servers, application servers, and similar tools that support application development and delivery. Middleware is the foundation of modern information technologies based on XML, SOAP, Web services, and service-oriented architecture.

2.3 Where is middleware?
Middleware sits “in the middle” between software applications that may be run on different operating systems. It is similar to the middle layer of a three-tier single system architecture, except that it is stretched across multiple systems or applications [2]. Examples include telecommunications software, transaction monitors and messaging-and-queueing software.

The distinction between operating system and middleware functionality is, to some extent, arbitrary. While core kernel functionalities can only be provided by the operating system itself, some functionalities previously provided by separately sold middleware is now integrated in operating systems. A typical example is the TCP/IP stack for telecommunications, nowadays included in almost every operating system.

Middleware could be imagined as a layer that lies between the application code and the run-time infrastructure. Middleware generally consists of a library of functions, and enables a number of applications to use these functions from the common library rather than re-create them for each application.

2.4 Why use middleware?
Middleware is useful for a number of reasons when designing large, distributed systems. Some relevant aspects are depicted in [1]:

Portability: middleware offers a common programming model across language and/or platform boundaries, as well as across distributed end systems. Thanks to this, it is possible to make cooperative applications developed in different languages (e.g. Java and C++) and executed on different operating systems (e.g. Windows and Linux) without any specific effort by the programmer.

Reliability: middleware are developed and tested separately from the final application. This allow the application programmer to abstract low-level aspects and use (and re-use) a well-tested library.

Managing complexity: low-level aspects could be managed by suitable libraries that abstract specific operating system or hardware aspects. This simplifies development and reduces the probability of errors.

2.5 Middlewares functionality and types
Middleware software can be divided by functionalities and target applications into three main categories: application-specific, information-exchange and management and support middleware.

Application-specific middleware provides services for various classes of applications such as distributed-database services, distributed-data/object-transaction processing and specialized services for mobile computing and multimedia.

Information-exchange middleware handles the exchange of information across a network. It’s used for tasks such as transferring data, issuing commands, receiving responses, checking status and resolving deadlocks.

Management and support middleware is responsible for locating resources, communicating with servers, handling security and failures and monitoring performance.

3 Middleware in robotics
3.1 Introduction
Robotic systems are usually complex systems built on many different hardware and software components, as sensors and actuators as well as planners and control algorithms.
In general, on each robot runs a software that is responsible for reading sensors data, extracting the needed informations from them, computing the sequence of actions to accomplish a given task and controlling the actuators to execute the actions. Using a custom approach, there will be a single monolithic application that will handle all these tasks, making code maintenance hard and preventing every form of code reuse and sharing between different projects.

Such a scenario, with many hardware and software components that needs to communicate and collaborate to reach a goal, is exactly where a middleware can help improving the organization, the maintainability and the efficiency of the code. The whole application can be structured into many little concern separated tasks, as "get a sensor reading", "extract features from some data", "drive the motors to some speed". Different components can exchange data using a common communication channel provided by the middleware, using interfaces that are consistent between different applications. In this way, it becomes really easy to share and reuse code among different projects, or change an algorithm to get some functionality as it is only necessary to keep the same interface. As an example, if you need to switch from a proximity sensor to another, it is possible to write a new component that share the same interface and update it without modifying the rest of the application. This concept can be extended to large and complex applications, in which using a middleware can clearly improve the overall code organization and reduce the programming effort.

3.2 Robotic middleware overview

There are many projects around claiming to release a middleware for robotics, all sharing almost the same basic concepts and goals, but many of them exists only as proposal and others are not being developed any more. In this work we focus on the projects that gained more popularity in the robotics community and that are still supported and actively maintained. For a more extended overview of middleware for robotics refer to [26].

3.3 OROCOS

The idea of starting a Free Software project for robot control has born in December 2000 on the mailing list of EURON, motivated by over two decades of rather disappointing experiences and failures in trying to use commercial robot control software for advanced robotics research [5]. At that time there were no open source general-purpose robot control software available, making OROCOS an innovative project that interested many people.

The project has been funded by European Commission with the participation of three partners: K.U.Leuven in Belgium, LAAS Toulouse in France and KTH Stockholm in Sweden.

3.3.1 Overview

The OROCOS project aims at becoming a general-purpose and open robot control software package [7]:

- under Open Source license like LGPL.
- with extreme modularity and flexibility such that a developer can build his system from scratch while other developers can contribute to the modules they are interested in, without the need to deal with the code of the whole system.
- of the highest quality from both the technical, documentation, and software engineering points of view.
- independent from (but if possible compatible with) commercial robot manufacturers, claiming that the OROCOS code should become an inevitable de facto Open Standard.
- for all sorts of robotic devices and computer platforms.
- featuring software components for kinematics, dynamics, planning, sensing and control, hardware interfacing, etc. Components are intended in the meaning of software objects that can dynamically be added or removed from a network and that offers its services through a neutral and programming language independent interface.
3.3.2 Organization

OROCOS is composed of 4 C++ libraries, shortly introduced in the next paragraphs.

The Oros Real-Time Toolkit (RTT) is not an application in itself, but it provides the infrastructure and the functionalities to build robotics applications in C++. The emphasis is on real-time, on-line interactive and component based applications.

The Real-Time Toolkit (RTT) library allows application designers to build highly configurable and interactive component-based real-time control applications.

The RTT allows components to run on real-time operating systems and offers real-time scripting capabilities, the component communication and distribution API (CORBA [6] based) and XML configuration (Fig. 2).

A RTT component can be written for example to control devices ranging from sensors to complete robots, to capture and plot a data flow, to tune an algorithm at run-time or to connect to a user interface.

Each component is an extension of the TaskContext primitive, an active object which offers thread-safe and efficient ports for (lock-free) data exchange, can react to events, process commands, or execute Finite State Machines in hard real-time.

Components can be configured on-line through a property interface (set/get values) and XML files.

The Oros Components Library (OCL) provides some ready to use components. All components are built on RTT, and some of them can use the KDL or the BFL libraries.

It is possible to find working components to interface with hardware devices, to handle path and task planning, to control various types of robot as well as to record and display data flows and to debug the whole application.

The Oros Kinematics and Dynamics Library (KDL) is a C++ library which allows to calculate kinematic chains in real-time. The Kinematics and Dynamics Library (KDL) develops an application independent framework for modeling and computation of kinematic chains, such as robots, biomechanical human models, computer-animated figures, machine tools, etc.

It provides class libraries for geometrical objects (point, frame, line,...), kinematic chains of various families (serial, humanoid, parallel, mobile,...), and their motion specification and interpolation.

The Oros Bayesian Filtering Library (BFL) provides an application independent framework for inference in Dynamic Bayesian Networks, i.e., recursive information processing and estimation algorithms based on Bayes rule, such as (Extended) Kalman Filters, Particle Filters (Sequential Monte methods), etc.

These algorithms can, for example, be run on top of the Realtime Services, or be used for estimation in Kinematics and Dynamics applications.

3.4 Orca

Orca is an open-source framework for developing component-based robotic systems. It provides the means for defining and developing the building-blocks which can be assembled together to form arbitrarily complex robotic systems, from single devices to distributed sensor networks.

3.4.1 Overview

Orca project grew out from OROCOS at KTH Stockholm in 2003. The main goal pointed out by its developers is the software reuse, that they define a key factor to continue progress in robotic research and industry.
Orca tries to achieve this goal focusing on three concepts:

**Enable** software reuse by defining a set of commonly-used interfaces.

**Simplify** software reuse by providing libraries with a high-level convenient API.

**Encourage** software reuse by maintaining a repository of components.

Orca define itself as an *unconstrained* component-based system, meaning that it wants to provide an infrastructure to build robot software but not to force the user to rewrite all his code from scratch. In this way, Orca adopts a Component-Based Software Engineering approach without applying any additional architectural constraint and uses a commercial open-source library for communication and interface definition.

Orca also provides some libraries for common applications and tools to simplify component development, but makes them strictly optional to maintain full access to the underlying communication engine and services.

The main difference between Orca and OROCOS lies in the toolkit on which the communication engine is built. Orca developers have replaced CORBA, developed since 1991 and used in OROCOS, with a modern framework from ZeroC[9]: Internet Communication Engine (ICE).

ICE developers hardly criticize the use of CORBA in nowadays projects, claiming that it is based on old assumptions, developed with old technologies and difficult to use [10].

Orca is deeply based on ICE, as can be noticed looking at its organization.

### 3.4.2 Organization

Orca is composed by some core services, inherited from ICE, that form the *infrastructure* and a set of common components, that can be used and extended to build custom applications.

The core services are:

**IceGrid registry** provides a Naming service: a mapping from logical interface names to physical addresses. It's currently the only way for components to find one another.

**IceStorm service** is an event service, used to decouple message publishers from subscribers. Typically, there is one IceStorm service per hosts.

The application is then built by components, that know each other through the IceGrid registry and communicate together using the IceStorm service.

Components can interface with hardware, using drivers and driver interfaces, that should be consistent in the same device family (e.g. laser scanners have different drivers but the same interface).

Orca developers provide some drivers for popular hardware, like cameras, laser scanners, robot platforms and manual controllers, as well as some components related to common tasks, like path and goal planners, computer vision algorithms, motion controller etc.

### 3.5 ROS

ROS [11] - Robot Open Source - is a recent initiative of Willow Garage [12], a research lab founded in late 2006 to accelerate the development of non-military robotics and advance open source robotics software.

As Willow Garage produces both software and hardware, ROS is tested on their robots through a set of *milestones* that show the progress of the development [13], [14].

#### 3.5.1 Overview

ROS is defined as an *open-source meta-operating system*, highlighting that it wants to be something more than a middleware. It provides the services expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. Its primary goal, as for Orca, is to enable code reuse in robotics research and development and to release ready to use software packages for a large set of common tasks.

ROS provides also a community website for sharing and collaboration, and repositories to store and distribute software packages.
3.5.2 Organization

ROS is organized in three layers:

- **Filesystem** level: ROS resources stored on disk.
- **Computational graph** level: a peer-to-peer network of ROS processes that are processing data together.
- **Community** level: ROS resources that enable separate communities to exchange software and knowledge.

The core of the system is the computational graph, that is described as a distributed network of processes, called *nodes*, individually designed and then loosely coupled at runtime. This network is composed by some basic elements:

- **Nodes** are processes that perform computation to absolve some task (e.g. read laser range-finder, control motors, perform localization, path planning...).
- **Master service** provides name registration and lookup to the rest of the Computation Graph.
- **Parameter server** allows data to be stored by key in a central location.
- **Messages** are used by nodes to communicate with each other. A message is a data structure, comprising typed fields.
- **Topics** are used to identify the content of a message. Messages are routed via a publish/subscribe transport system. A node sends out a message by publishing it to a topic, and another node reads it by subscribing to that topic.
- **Services** are used when the public/subscriber model is not appropriate, as for request/reply interactions. A providing node offers a service, that is defined as a pair of request and reply structures. A client node sends the request and waits the response.
- **Bags** are format for saving and playing back ROS message data.

The ROS Master acts as a name service in the ROS Computation Graph. It stores topics and services registration information for ROS nodes. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. The most common protocol used in a ROS is called TCPROS, which uses standard TCP/IP sockets. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run. Nodes connect to other nodes directly, as the Master only provides lookup information. Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed connection protocol.

This architecture allows for decoupled operation, where the names are the primary means by which larger and more complex systems can be built. For example, every sensor node publish scans, without knowledge of whether anyone is subscribed. All the filter nodes subscribe to scans, without knowledge of whether anyone is publishing them. The nodes can be started, killed, and restarted, in any order, without inducing any error conditions.

Names have a very important role in ROS: nodes, topics, services, and parameters all have names. Every ROS client library supports command-line remapping of names, which means an compiled program can be reconfigured at runtime to operate in a different Computation Graph topology.

ROS developers distribute a large amount of ready to use components, like hardware drivers for many devices and algorithms for a wide set of tasks. ROS is also distributed as bootable dvd-image, built on Ubuntu GNU/Linux, that can be installed on any computer to get a complete ROS environment in minutes.

3.6 BRICS

BRICS [15] is a joint research project funded by the European Commission that involves a large group of partners (Fig. 3). BRICS initiative wants to identify and document best practices in the development of complex robotics systems, to refactor existing components in order
to achieve a much higher level of reusability and robustness, and to support the robot development process with a well-structured tool chain and a repository of reusable, configurable code[17].

The idea from which BRICS has born is to extend the formalisms of software development process used to develop large software systems to the robotics world, as the robot development is by far not as well defined and established as a structured process.

3.6.1 Overview
The prime objective of BRICS is to structure and formalize the robot development process itself and to provide development tools, computational models, and functional libraries, which allow engineers and developers of complex robotic systems to reduce the development time and effort by an order of magnitude. In detail, BRICS sets the following objectives:

Survey and analyse completed robot developments with respect to their deficiencies in terms of tool support, reuse of components, use of harmonized interfaces, and use of functional and model libraries.

Promote and moderate a community-wide discussion on the identification, evaluation and refactoring of best practice in robotics algorithms and software systems.

Design and implement an integrated development environment for robotics (called BRIDE) and an accompanying software repository of best practice robotics algorithms (called BROCRE).

Significantly promote the interoperability of hardware and software components by harmonizing the interfaces and as well as the communication between these components,

Define and implement showcases which allow to measure and evaluate the progress between today practices and the BRICS enabled robot development process.

3.6.2 Organization
BRICS project focuses on all the robot development process, that can be splitted in a set of different research topics.

Hardware building blocks: after a fundamental analysis of the hardware requirements in robotics research and industry, BRICS partners will develop robotics platforms, components and interfaces, to define and implement showcases which will allow to study and measure the acceleration of the robot development process. The implemented hardware will promote the interoperability of hardware and software components by harmonizing the interfaces and as well as the communication and data exchange between these components.

Architectures, middleware, and interfaces: this research topic aims at identifying complete robot control architectures or architectural subsystems and patterns, which are attractive for reuse by researchers and application developers.

The first part of this topic will focus on the identification of best practice in architecture, middleware, and interfaces analyzing the ideas and concepts produced by previous researches, trying to make these specific works more reusable.

The second objective is the development of a software platform for robotics, that has to deal with the heterogeneity of hardware devices and the complexity of communication and distributed systems. The result will be a middleware for the robotics domain that should provide special support for handling soft and
hard real-time requirements, communication facilities not supported by current middleware (CAN-bus etc.), and for advanced failure detection and failure handling capabilities to achieve robust autonomy.

The last part of this topic regards the development of functional/control architecture workbench for robotics, that should provide configurable blueprints for well-known robot control architectures, which have been refactored to exploit the robust, state-of-the-art software technologies.

**Best practice in robot algorithms:** BRICS wants to harmonize robotics algorithms [16], so that they are easily interchangeable, by developing a framework that can be applied to a large number of robotic tasks and algorithms, and not only to one or two. This activity will address different scientific and technical key issues, like the harmonization of operational context, conditions and performance requirements for algorithms, as well as the harmonization of data structures and interfaces and the definition of common test conditions.

**Model driven engineering and toolchain:** this activity will focus on creating the Model-Driven Engineering (MDE) models for robust autonomy in robotic systems, identifying what MDE models are appropriate abstractions of the complex, autonomous and robust multi-agent systems that are so typical for intelligent robot systems [18].

It is also planned to create a BRICS Integrated Development Environment, based on Eclipse, that will provide to the robotics community a Model-Driven Engineering toolchain that contains the robotics domain specific models and tools to build BRICS components and interfaces.

**Openness and flexibility:** the objective of this research topic is to provide the conceptual and technological means that will allow other researchers to develop open and flexible robotic software. BRICS researchers want to identify recurrent design problems and define reusable design solutions (design patterns) and to define metrics to assess the software quality.

**Robust autonomy:** Robust Autonomy (RA) is an ability of the system to react on adverse and unexpected situations in both the environment and the hardware/software system in order to complete a given task in the best possible way. BRICS will contribute providing design principles, implementation guidelines, evaluation criteria and use case implementations for robust autonomy.

4 Toward a Mixed HW/SW Middleware

Looking at the middleware frameworks we have presented, it is evident that all of them originated from computer science ideas, trying to extend those concepts to robotics. What was not plenty accounted, in our opinion, is that a robot is composed by many small hardware devices, with low computational power and limited connectivity, that can’t run complex software like the most of the middleware around. In fact there is no middleware that provide a way to connect pure software components and those software parts that are embedded in devices firmware.

4.1 Importance of hardware

A robot can be seen as a distributed system of embedded devices, like sensors and actuators, that need to interface each other by some communication hardware. Each device, generally, has some logic on board to handle its tasks, like a microcontroller or some programmable logic. This kind of embedded controllers have limited resources (e.g. some Kb of RAM and memory, 8-bit integer core, slow communication interfaces), that are not able to run a complex communication stack like CORBA or ICE, widely used by the majority of middleware for robotics. In our vision of a robotic system, there is the need to deeply integrate hardware and software, as the components of the complete architecture are not only software snippets (e.g. Orocos components or ROS nodes) but also real devices, that must fuse with the rest of the system. In this scenario, a middleware for robotics should be able to run also on embedded devices, using a simple communication protocol to share data and tasks between complex software applications and small hardware components.
4.2 Architectural proposal - AIRCan

AIRCan is a middleware proposal for electronic boards and small microprocessors. It was designed at Politecnico di Milano by AIRLab research group and formalized and prototyped in [19]. AIRCan will be briefly described in the following sections.

4.2.1 Architecture and components

AIRCan allows the communication among different devices relying on standard CAN (Controller Area Network) bus. CAN is the leading serial bus for embedded control. It was developed in the early 1980s, internationally standardized in 1993 [20] and recently revised [21] [22] [23] [24] [25]. The only constraint for the AIRCan architecture is that each component (which is ideally an electronic board with a microprocessor) must have a programmable CAN interface.

An AIRCan architecture provides two kinds of devices: generic devices and coordinator devices.

An AIRCan network has to be composed by a unique coordinator and one or more generic devices that works together in a service oriented paradigm. Each device exposes some services and needs other services to work properly. AIRCan is fundamentally a publish/subscribe architecture: the coordinator device knows, thanks to a bootstrap step, the association between devices and exposed services and between devices and needed services.

We can consider a simple example: suppose that you want to allow the control of motion of a mobile robot with two different electronic boards: the first one is the motor control board and the second one is a board that reads wheel encoders. It is clear that we need to connect these two boards and allow communication between them. In the AIRCan architecture, the odometry board publish a service that informs about encoder status and motor board needs this service. When the entire system starts, the coordinator manages published services and subscriptions and inform motor control board to use the service provided by the odometry board. After this initial phase, the two boards can work together exchanging messages with the needed and provided data.

The coordinator device has two other fundamental roles:

- Manage critical situations, like fault in other devices.
- Works as a gateway between the AIRCan network and a standard computer via USB connection.

The last point has a great relevance in AIRCan structure: since that the coordinator plays a gateway role between the hardware network of devices and a standard computer, it is possible to use the same paradigm of communication, based on publish/subscribe, between devices and pure software services. In the previously described example, we can imagine that if we want to test a newer control method for the robot motion control, we can develop and test it with a software algorithm running on the computer that expose a service for motor controls. When the algorithm will be stable, we can easily move this service on a dedicated board (e.g. the same of motor control) without any change in software and firmware of other devices.

4.2.2 Protocol specification

The AIRCan protocol mimics the TCP/IP stack. In particular we can consider that level one (physical layer) is provided by the specific CAN module on microprocessor. Layer two (transport) manages input and output buffers and allow to take a single frame from the receiving buffer and to queue a single frame on the transmission buffer.

Layer three has the function of fragmentation and recomposition of messages and it is the interface for the application programmer. Thanks to this it is possible for an application programmer to develop a microprocessor firmware based on AIRCan specifications that communicate with other boards or computer software via messages of arbitrary length.

As it is impossible to describe here all AIRCan details, refer to [19] for further informations.

4.2.3 AIRCan prototype and future works

AIRCan was prototyped and tested on PIC microprocessors (both gateway and devices)
and a complete library was developed in C language. Future works include the porting of the library on other platforms (e.g.: ARM microprocessors) and the application of a complete AIRCan structure to a real robot architecture.

5 CONCLUSION

Having explored the state-of-the-art of middleware frameworks for robotics we can notice that there is a lot of activity in this research field, as the need of a more structured approach developing software for robots is widely acknowledged. Many of the reviewed projects share the same basic concepts and goals, while BRICS wants to join the best aspects from all of them in a single framework. In our opinion the biggest lack in the available frameworks is the vision of the robotic system as a set of independent devices connected to a central processing unit instead of a distributed system of cooperating devices that share data and services to reach a common goal. Our expectation is that in the near future middleware for robotics will be developed taking into account the cooperative and heterogeneous architecture of a real robot discarding the high-level approach of treating a robot like a form of software.

REFERENCES